Planlet: supporting plan based user assistance

Gary Look, Intel Corp.
Joao Sousa, Intel Corp.
Miryung Kim, University of Washington

IRS-TR-03-005

April, 2003

Intel Research

Planlet:
supporting plan-based user assistance

Miryung Kim[1]
Department of Computer Science and Engineering, University of Washington
miryung@cs.washington.edu

Gary Look
Artificial Intelligence Lab, Massachusetts Institute of Technology
garyl@ai.mit.edu

João Pedro Sousa
Intel Research Seattle
School of Computer Science, Carnegie Mellon University
jpsousa@cs.cmu.edu

Abstract. The Ubiquitous Computing revolution brings with it the pressing need of relieving users from routine chores and of assisting users in carrying out complex tasks. Increasingly, software developers recognize the potential of exploiting knowledge about user plans to enable computer systems to play such roles. Specifically, plan-aware computer systems can proactively remind the user to perform planned task steps; they can automatically configure the user's working environment; and they can guide the user in following best-known work practices.

We describe Planlet, a generic software layer for representing plans of user's tasks. Planlet makes it easier to build task-aware Ubiquitous Computing applications by providing generic services to hold and manipulate knowledge about user plans, habits and needs.

---

[1] This is a report of collaborative work that took place during the summer of 2002 at Intel Research Seattle and the Cell Systems Initiative of the University of Washington. Authors are listed in no particular order.

1 Traveling to Planlet

We are in the midst of a wave of change – one where computers will become aware of users' tasks. Looking back, people had to be aware of the task of using a computer in addition to the actual task they are using the computer to perform. Like programming, computer configuration and even usage required training, or at least deliberate acquisition of skills. But computers are increasingly pervading all aspects of human life, and that dictates a change in the roles of computers and of humans. Using computers can no longer be more complex or intrusive than using a kitchen appliance: what you really want is to cook dinner. What makes it easy to use a kitchen appliance? It is carefully designed to serve a particular task – in that sense, the appliance embeds knowledge of the user's intent and of the task for realizing that intent.

A partial solution for the problem is offered by the emergence of task-specific computing gadgets. No doubt many such gadgets will be developed and play a valuable role in making life easier. However, such gadgets will only take us so far, in the same way that cooking a gourmet dinner is a far more complex task than using a blender: it requires knowledge specific to the occasion, the tastes of the guests, budget, available ingredients, the time available for preparation, etc. Once these parameters are known, a task-aware computer could proactively assist even an inexperienced cook in following through the preparation steps: offering guidance, issuing reminders, or even actuating devices such as pre-heating the oven on behalf of the user.

Many domains of human activity involve following through sophisticated task plans: conducting biology experiments in a lab [3], repairing complex mechanisms [9] and even mundane tasks like preparing a talk [4] or cooking dinner [7].

Planlet is a generic software layer for representing plans of user's tasks. Planlet makes it easier to build task-aware Ubiquitous Computing (Ubicomp) applications by providing generic services to hold and manipulate knowledge about user plans, habits and needs. The knowledge embedded in Planlet can be used by Ubicomp applications to reduce the task overhead and user distractions. For instance, applications can proactively remind the user to perform planned task steps; they can automatically configure the user's working environment; and they can guide the user in following best-known practices.

There are fundamental assumptions in the domain targeted by Planlet that set it apart from research on planning and research that uses planning. First, in Planlet's domain, the agent carrying out a plan is human – not a computer. Second, the role that Planlet supports is to assist, rather than control the user. Of course, the actions of the user can be influenced by the contents of the plan, but are ultimately determined by the user's will and his interaction with the physical context. In that sense, a plan captures what the user *can* or *should* do, not what he *will* or *must* do. This has profound consequences to how Ubicomp applications exploit knowledge about the user's plan, and more broadly to the roles played by users and by computer systems based on Planlet.

Section 2 elaborates on the goals and assumptions of Planlet and describes how Ubicomp applications can exploit the knowledge represented in Planlet. Section 3 compares Planlet with related research. Section 4 discusses the high-level architecture of Planlet and how it interacts with Ubicomp applications. Section 5 describes the constructs available for parametric plan definition, how plan definitions are instantiated, and describes

the process of carrying out a plan. Section 6 describes an implementation of Planlet in Java, focusing on the most important aspects for the writers of Ubicomp applications on top of Planlet.

2 What is Planlet like

Where do plans come from

Planlet assumes that the plan is given – i.e., it relies on an outside source, such as the Ubicomp application itself, to obtain the plan. Depending on the domain, that outside source may construct a plan given a goal (AI planning); it may learn plans by observation of behavior patterns (machine learning); or it may rely on a user who is a domain expert to define the plan (plan editing). Planlet is not concerned with *how* the plan definition is obtained – it exposes an API that enables external components to create and modify a plan.

Who carries out plans

Planlet assumes that plans are carried out by agents external to Planlet. Planlet assumes that these agents are humans over which Planlet – or any Ubicomp application built on top of it – has ultimately no control of. Furthermore, Planlet makes no assumptions about the working capacity of the agents (one person may be able to perform two steps simultaneously while another would rather do them in some order) or even *multiplicity* (one person may ask for help from others when facing a hard task, or he may ask for help to speed up a repetitive task).[2]

What does a plan look like

A plan is given by a directed acyclic graph, where the nodes are either subplans or atomic steps and the arcs represent the flow of materials between nodes.[3] The term *material* denotes whatever is produced and consumed by plans. Each domain defines its own vocabulary of atomic steps – Planlet exposes an API to define such steps. Both plans and steps define a set of inputs and a set of outputs: directed flows between outputs and inputs define the plan's graph. Conditional flow is obtained by associating conditions to the arcs in the plan graph: a material only flows along the arc if the condition holds.

Planlet distinguishes the notions of plan definition (or template) and plan instance. Plan definitions are parametric and can be instantiated many times. For instance a plan definition for cooking breakfast may be instantiated every morning by some user, and each time for some number of servings. Plan instances get to be carried out.[4]

As a user makes progress through a plan, the nodes in the plan graph are marked up as done. When a node is marked as done, all its output materials become available to the

---

[2] Planlet cannot make simplifications to the plan based on a limit to the number of activities the agent can carry out simultaneously or based on prescribing an arbitrary ordering.

[3] Iteration is supported by recursion. Section 5 elaborates on this.

[4] When for simplicity we say that a plan is carried out, it should be understood that we are referring to a plan instance.

consumer nodes downstream. When a step has all its input materials available, it is included in the list of steps Planlet expects the user to perform next. Planlet supplies the Ubicomp application with that list so that the application can be proactive in assisting the user with to carrying out his plan.

What is manipulated in plans

When a user carries out a plan, he possibly manipulates information inside the computer system and most importantly, he manipulates physical materials: chemicals, tools, documents, groceries, etc. This fact entails two important consequences.

The first consequence is that Planlet does not know for sure when or whether steps have been completed. This stems from the acknowledgement that there is, at best, limited automatic sensing of the physical world. For instance, when a plan indicates that the user should "get the screwdriver", Ubicomp applications may have to rely on the user to confirm when and whether he fulfilled the planned step. By the same token, pre and post conditions to actions may have to be considered merely *informative* to the user. Suppose, that the planned step is to "fry eggs" with "use a clean frying pan" as a precondition and "the eggs are now fried" as a post condition. There is really not much Planlet or any Ubicomp application can do if the user decides to use a greasy pan, or doesn't cook the eggs thoroughly. In more structured domains, like a heavily instrumented biology lab, the Ubicomp application can use sensing in the environment to determine whether or not the step was completed as prescribed and can take appropriate actions if not. Planlet leaves such monitoring and decisions to the application.[5]

The second consequence is that the list of expected steps is not only dependent on the execution state of the plan, but also on the current physical context around the user. This stems from the fact that accessing materials is strongly constrained by the physical context: location, obstacles, weather, etc. For instance, suppose that during the course of repairing the roof, the user can either get the screwdriver – from the toolbox in the basement – or continue to nail tiles to the roof boards already in place. If sensors can locate the user in the basement, then the action of continuing to nail tiles on the roof is no longer expected, regardless of being possible according to the plan graph. Furthermore, another action, say get paint and a brush from the shelves in the basement may become more likely by context affinity. A Ubicomp application can save the user many distractions by recognizing the affinity between steps that derives from shared context. In the example, the application can issue a reminder for the user to get the paint in addition to the screwdriver, saving him another trip to the basement later on. The current version of Planlet supports annotating context to a step – as markup text – but contains no features to explore that information, sorting out step affinity or relative likelihood given the current user context. Those features are planned for future work.

---

[5] In other words, Planlet does not assume or rely on a 100% up-to-date replica of the state of materials in the real world. Planlet does keep notes of which state the materials *should* be in, according to the plan, and informs the application/user of that. The fact that Planlet considers pre and post conditions as informative, or more accurately as black boxes, is orthogonal to the use that the application makes of such pre and post conditions. When the application defines the plan, it can associate conditions that have a precise meaning for the application, and that can be opportunistically evaluated by the application.

Carrying out plans

The assumption underlying Planlet is that the role of Ubicomp applications is to *assist* rather than *control* the user in carrying out a plan. A plan and the progress through it represent knowledge about events in the real world. These events are beyond the control of and, in many cases, beyond accurate monitoring by computer systems. When automatic context sensing is limited, how much Planlet knows about the progress through the plan is very much dependent on the good will of the user in letting Planlet know. Therefore, Planlet takes a liberal, accommodating attitude towards tracking the plan.

Nevertheless, Ubicomp applications can take a range of measures upon noticing that the user is deviating from the plan. Those measures may range from making sure the user is aware he is deviating (and allow or even register the new user actions as an alternative branch in a known plan), to refusing notice of completion of other than the permitted steps (akin to workflow systems), to actuation in the physical environment or communication with authorities whenever safety may be compromised (e.g. when monitoring Alzheimer's patients). Planlet supports this range of actions by offering the application two types of mechanisms.

First, Planlet offers two modes of marking a step as done: soft complete and hard complete. When soft complete is called on a step or subplan, Planlet checks if that step is on the list of steps that the user is expected to take next. If not, it returns a diagnostic and does not change the state of the plan. Such diagnostic is examined by the application and an appropriate measure can be taken. If the application is able to double check that, from the point of view of the user and/or user's safety, it is ok to enforce the completion notice, it can call hard complete on that same step or subplan. For a hard complete, Planlet assumes that the user did not bother to fill in all the details of what he is doing – and that the application is ok with that – so it just marks the step as being done (albeit done unexpectedly because it is a step that is downstream from an expected step) and advances the plan execution state to the indicated point.

Second, Planlet allows parametric plan definitions to be instantiated incrementally at runtime, as the required information becomes available. Planlet also allows the plan to be changed at runtime, by inserting or eliminating steps (before they are marked as completed).[6] To which extent this mechanism is used is a decision of the application and derives from how constrained the domain is. One can imagine giving the user a lot more leeway when cooking dinner than when repairing a high-voltage mechanism.

3 Planlet and the others

There are two broad research categories related to plans: one is concerned with deriving plans, the other with exploiting plans. The first category, known as *planning*, is widely explored in the field of Artificial Intelligence [10,11]. The research underlying Planlet fits into the second category: given a plan, how can it be exploited for the advantage of the user or his organization. Examples of research in this category range from optimizers

[6] Although not supported in the API currently available, this feature is part of Planlet's design and will be supported soon.

in programming language compilation [1,6], to a broad range of workflow-based systems [2,5,8], to PERT charts used in project management.

Most plan models (e.g. [2,5,8]) adopt a control flow perspective that is explicit about ordering constraints (for instance: do *A* then *B* then *C*) but implicit about the *reasons* for those constraints. Like Data-Flow Graphs (DFGs), Planlet adopts a pure material flow model to derive the constraints on plan execution: two steps *A* and *B* are constrained to occur one after the other only if B consumes something that originates from what was produced by *A*. Consequently, Planlet's model is explicit about the sets of input and output materials corresponding to deliverables of the project's activities. It is also explicit about the type of material – either data or a physical material – and handles propagation and computation of data in the usual way.[7]

There are two benefits in forcing the plan to be explicit about the reasons for sequentiality, which derived from material producer-consumer relationships. First, it maximizes the potential for identifying the next possible steps since there are no artificial sequentiality constraints based on *how* the planner *thinks* the user will carry out the plan. By doing this, we make it easier to take advantage of probabilistic models built on top of the material-based constraints. These models can be built to reflect user habits or best-known practices (future work). By being able to discern the most likely steps among the possible next steps, Ubicomp applications can offer added-value assistance to the user.

Second, whenever physical materials are produced and or consumed by a task, the physical context (location etc.) poses severe constraints to the feasibility of carrying out a step: if a required material is across town, the user is not likely to carry out the step no matter being possible according to the plan. Exposing materials and their constraints as part of the plan is critical to recognize this kind of situation.

Similar to DFGs used in programming language compilation, Planlet adopts a producer-consumer perspective of plans: as soon as a piece of data (material) is produced, it is made available to all its consumers; and a computation (plan step) can be carried out as soon as all its inputs are available and no earlier. However, the goal of compilers in exploiting DFGs is to optimize the order of computations (the plan), given a well-known execution capacity and strategy of the executing agent – the CPU.

By contrast, the goal of Planlet is to exploit knowledge of the plan to assist the user. By making no assumptions on the execution strategy or capacity of the user, Planlet exposes massive concurrency of plan steps – there are no ordering constraints other than those derived from availability of materials. During plan execution, whether or not actual parallelism occurs is out of the control of Planlet: parallelism is spontaneous rather than precisely planned. This is a legitimate assumption since the goal of Planlet is not to control the execution of the plan but to predict which are the steps that the user may be taking next (steps that are *enabled* according to the flow graph semantics) so that Ubicomp applications can proactively assist the user.

---

[7] Currently Planlet supports numbers and strings as data types and treats the physical materials type as a single bit with the semantics of the material being available.

The goal of workflow systems and of PERT charts is to guide the behavior of users. This guidance often takes a compulsory flavor, but we see no reason why Planlet couldn't be used as a base for a workflow or project-tracking tool, sufficing that the application on top of Planlet takes the more conservative interpretation of Planlet's warnings and disallows deviations from the plan. Aspects of plan recognition, context sensing and interaction with the user are a major focus of systems like [2,8]. Such aspects are specific to application domains and, as such, are left out of Planlet.

Table 1 summarizes the key assumptions of Planlet, their consequences, and how those are distinct from the ones in related research. As discussed in Sections 2 and 3, the key assumptions concern which agent carries out the plan and the ensuing control strategy; the goal of representing the plan, and the constraints on accessing the materials manipulated during the course of plan execution.

	Planlet	Compiling	AI Planning
Executing Agent	user	computer	computer
Assumption	no control over executing agent (user) - user may deviate from known plan	compliant deterministic agent	compliant or biddable deterministic agent
Control	• non-deterministic • driven by will of user • agent has unknown multiplicity ("*hey Bob, give me a hand*") • parallelism is spontaneous	• deterministic • parallelism explicitly and precisely planned	• deterministic • parallelism explicitly and precisely planned
Assumption	plan is given	plan (program) is given	objective is given
Goal of computing system	• proactively assist the user to carry out the plan • record execution trace	transform plan to optimize execution	generate optimal plan
Assumption	task manipulates information as well as physical materials.	task manipulates information	task manipulates information as well as physical materials usually in a limited context.
Access to materials	strongly constrained by physical context	unconstrained access (random access memory)	easy access (e.g. wood blocks on a table)

Table 1 Planlet and the others

4 Building on Planlet

As mentioned in Section 2, Planlet is interested in three aspects of a plan: the plan definition, the plan instance, and the plan's execution. This section describes the high-level interaction between an application and Planlet during plan definition, instantiation, and execution. This section also introduces the design components used to create a plan and briefly describes the functionality that each of these components support. A more detailed description of the APIs is provided in Section 6.

In the plan definition stage, some form of plan editor (which could be an AI planner, a plan recognition algorithm, or the user himself) creates a plan by identifying the subplans that should be performed and lays out the producer-consumer relationships between these subplans. A partial ordering of the subplans is thus determined in this stage. The plan editor then tells Planlet what this plan is through a set of Planlet plan definition APIs. In

Planlet, plans can be defined with hierarchy, conditional execution of subplans, and iteration of subplans.

At definition time, plans may be defined parametrically, with unbound parameters such as resource assignments and deadlines. These parameters are fixed in Planlet's internal plan model by the application at instantiation time. Plan instantiation may be interleaved with the third stage, plan execution.

During plan execution, the application can ask Planlet to use its knowledge of the user's plan to inform the application of the things a user could be expected to do next. The application then uses this information to interact with the user in an appropriate way, for example, to reserve resources the user may soon use. Exactly how this information is used is a decision left entirely to the application.

As the user completes parts of his plan, the application passes this information along to Planlet. Planlet then updates its internal model of the user's plan and what things the user could be expected to do next. This interaction between Planlet and a plan-aware ubiquitous computing application is illustrated in Figure 1.

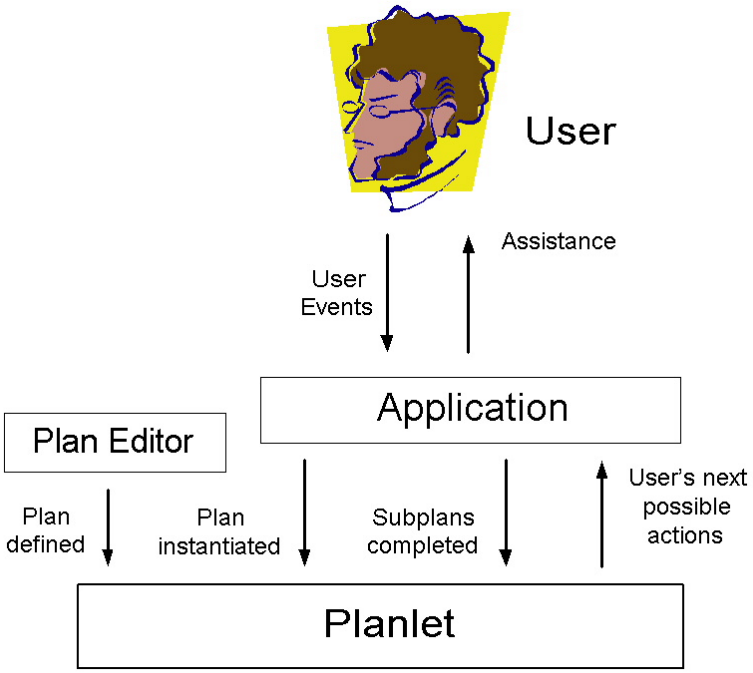

Figure 1. Some form of plan editor (which could be an AI planner, a plan recognition algorithm, or the user himself) first defines user plans. The application then instantiates the plan, fixing such plan properties as resource assignments. Then, as the user executes his plan, Planlet informs the application of what the user could do next so that the application can take appropriate action.

Planlet provides six kinds of constructs for use in defining plans. The first four types of constructs are all types of plans; these constructs are the Step, the HierarchicalPlan, the LoopPlan, and the MultiPlan. The other two types of constructs are types of flows; these two constructs are the Flow and the MultiFlow.

The Step represents an atomic action. A HierarchicalPlan is used to represent a collection of plans that are grouped together in a hierarchy of plans. The LoopPlan is a construct that is useful for representing a plan that is repeated some number of times, and a MultiPlan is a construct that is useful for representing multiple, independent copies of a given plan that need to be performed. The Flow and MultiFlow constructs are used to represent the flow of materials between plans.

How plan editors use these constructs to define plans is explained in the next section. Before doing that, however, we discuss the functionality (i.e., APIs) that each of these components provide to the programmer.

All of the four plan constructs, Step, HierarchicalPlan, LoopPlan and MultiPlan share the APIs described in Figure 2.

```
get/setProperty()       // gets or sets a particular parameter of the plan

get/setPlanName()

get/setContext()        // gets or sets any context information (currently a
                        // string) associated with this plan

getNumInputs/Outputs()  // return the number of inputs this plan
                        // requires or the number of outputs produced

get/setInputType( or OutputType)
                        // inputs and outputs to a plan are
                        // either physical materials or data

get/setInputName( or OutputName)
                        // inputs and outputs can be
                        // referred to by name

instantiate()           // turns the plan definition into a corresponding
                        // instance, fixing plan properties in the process

toString()
```

Figure 2. APIs common to all Planlet plan constructs.

In addition to these APIs, the Step construct has the following API:

```
setNumInputs/Outputs()      // change the number of inputs required or
                            // outputs produced by this Step
```

The HierarchicalPlan also has APIs in addition to the ones described above. These APIs are described in Figure 3. Once a plan is instantiated, the plan instance provides the API in Figure 4 for use in updating and querying Planlet's model of the user's progress through his plan. Lastly, Flows and MultiFlows share the API in Figure 5.

```
setNumInputs/Outputs()      // change the number of inputs required or
                            // outputs produced by this HierarchicalPlan

add/removeSubplan(Plan)

add/removeFlow(Flow)        // add or remove a flow between two subplans in
                            // this HierarchicalPlan
```

Figure 3. In addition to the APIs all plans have, a HierarchicalPlan also has the above APIs.

```
getPossibleActions()        // return a list of all Steps the user could,
                            // according to the plan, possibly do next

softComplete()              // if this part of the plan has produced its
                            // outputs, then mark it as done; else, return
                            // a diagnostic indicating what needs to be done
                            // before this can be marked done as done.
                            // Update other parts of the plan as necessary

hardComplete()              // unconditionally mark this part of the plan as
                            // done.  Update other parts of the plan as
                            // necessary.
```

Figure 4. The APIs used to update and query Planlet's model of the user's progress through his plan.

```
getProducerPlanName()       // get the name of the plan that we are routing
getConsumerPlanName()       // material from (producer) or to (consumer)

getProducerPlanNum()        // it may be the case that a producer or
getConsumerPlanNum()        // consumer plan may have more than one input or
                            // output, and this identifies the particular
                            // input or output of the producer or consumer
                            // plan
```

Figure 5. The APIs exposed by the Flow and MultiFlow constructs.

5 Defining and using Plans

Planlet provides 6 kinds of plan definition constructs. A plan definition in the Planlet consists of 4 different kinds of plans and 2 kinds of conditional flows that define a producer-consumer relationship. In Section 5.1, we discuss the semantics of a plan and composition of a plan with emphasis on what the different types of plans all have in common. In Section 5.2, we present the four different kinds of plans. The distinct semantics of a step, a hierarchical plan, a loop plan, and a multi-plan are discussed. We define an expansion of a plan as a transformation from one of the Planlet convenience constructs (LoopPlan or MultiPlan) to a HierarchicalPlan. We discuss how plans are expanded at instantiation time or execution time.

In Section 5.3, a flow that defines a producer-consumer relationship is introduced. In addition, manifestations of disjunction, conjunction, and exclusive disjunction are shown. In Section 5.4, we present how a multi-flow is used to simplify multiple relationship patterns. Section 5.5 introduces the execution states of a plan. When a plan is updated with progress information, its state is updated. A plan instance can be in one of four possible states. Section 5.6 shows an example scenario of preparing a bacon and egg breakfast.

This example shows how Planlet uses plans in the plan definition, plan instantiation, and plan execution stages.

5.1 Commonality among different types of plans

Every plan construct in Planlet is defined with inputs and outputs. In addition plans are annotated with property expressions and context information that can be refined by a plan definer. The property expressions can be evaluated during execution of a plan. The context information is associated with a plan to denote physical context information, such as time, location, and user. As we mentioned in Section 2, the context information can be understood as one of properties that Planlet uses to specially filter and determine next possible actions. Graphically, a plan is represented as a box with several input ports and output ports. (Refer to Figure 6)

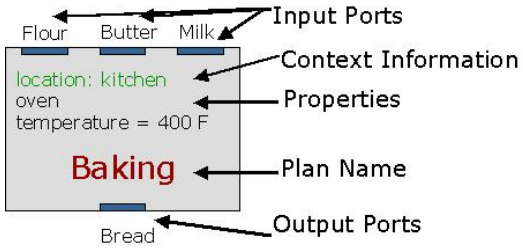

Figure 6 Example of Plan definition.

5.2 Different types of a Plan and their expansion

In this section, we introduce 4 kinds of plan definitions and their expansions.

5.2.1 Step

A step in a plan definition is an atomic action that takes inputs, executes an operation on these inputs, and produces outputs. In a step, an evaluated property can be produced as an output.

For example, in Figure 7, an addition operation is described as a step that takes two inputs A and B and produces C with the sum of A and B.

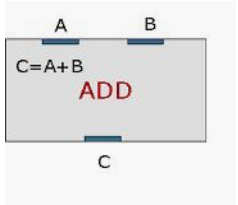

Figure 7 Example of Step (addition of two inputs).

5.2.2 Hierarchical Plan

A hierarchical plan contains a collection of related sub plans. Inputs to the sub plans contained in the hierarchical plan should be defined either as an input to the hierarchical plan or as an output of a sub plan contained in the hierarchical plan. Although a hierarchical plan may have associated property expressions or context information, a hierarchical plan does not perform operation on inputs by itself. Operations are performed in the sub plans

that compose a hierarchical plan. For example, Figure 8 represents a laundering example. Laundering consists of washing and drying laundry. The output of the first step, washing is used as an input to the second step, drying.

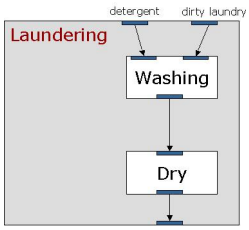

Figure 8 Example of Hierarchical Plan.

5.2.3 Loop Plan

A loop plan is a construct to denote repetition bound on a certain condition. It's similar to the while construct in a programming language. A loop plan is defined with a sub plan that is to be repeated and a condition expression that determines when the iteration terminates. The condition is evaluated when the necessary inputs are ready. When the condition is evaluated, a loop plan expands to a hierarchical plan with the same input information, property information, context information and output information. An expansion of a loop plan can occur at the initial instantiation time or at the execution time. When the condition is evaluated as true, this hierarchical plan contains the instances of a sub plan followed by the original definition of a loop plan with a different plan name. Figure 9 illustrates this process. When the condition is evaluated as false, the hierarchical plan does not contain any sub plans and simply routes its inputs to its outputs. In short, a loop plan represents iteration through recursive definition.

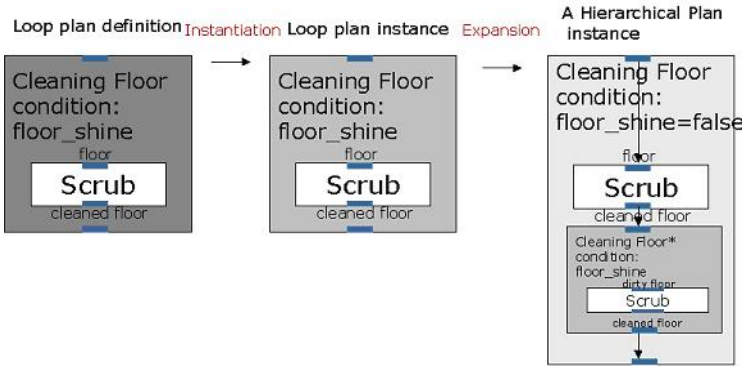

Figure 9 Example of Loop Plan definition, Instantiation of a Loop Plan, Expansion to a Hierarchical Plan.

The sub plan contained in a loop plan has the same cardinality of inputs and outputs and this cardinality is the same as the number of inputs and outputs in the loop plan. This design decision is made to define a repetition recursively without complication.

5.2.4 Multi-Plan

A multi-plan is a convenience construct used to denote multiple identical plans. It's similar to collection construct in programming language. One usage example of a multi-plan is to represent an operation that will be performed on a batch of samples in a biology lab.

A multi-plan is defined with a sub plan that is to be replicated and the expression that is to be evaluated as the number of replications. The expression is evaluated when the necessary inputs are ready. When the number of replication is determined, a multi-plan expands to a hierarchical plan with the same input information, property information, context information, and output information. This hierarchical plan contains the determined number of a subplan instances. The number of replications is evaluated and fixed only once, either at initial instantiation time or at execution time. (See Figure 10.)

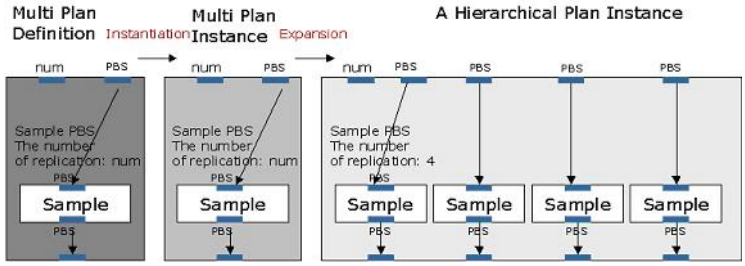

Figure 10 Example of MultiPlan definition, Instantiation of a MultiPlan, Expansion to a Hierarchical Plan.

The changes made during transformation (a loop plan to a hierarchical plan or a multi-plan to a hierarchical plan) are local in the sense that they do not affect other plan definition or instances.

5.3 Semantics of a Flow

A flow is a construct that defines a producer-consumer relationship between plans. A flow is identified by a producer plan, a port of the producer plan, a consumer plan, and a port of the consumer plan. There are 4 kinds of flow connection. The first case is the connection from an output port of a producer to an input port of a consumer. This flow is defined between steps or hierarchical plans. The second case is the connection from an input port of a producer to an input port of a consumer. The producer of this flow is always a hierarchical plan, and this flow is used to re-route the input of a hierarchical plan to the input of sub plans. The third case is the connection from an output port of a producer to an output port of a consumer. The consumer of this flow is always a hierarchical plan. This flow is used to re-route the output of a producer to the output of a producer's container plan. Finally the fourth case is the direct connection from an input port of a hierarchical plan to its corresponding output port. This is used in the base case of a loop plan expansion where inputs are not touched and are simply re-routed as outputs.

In Planlet, it is possible to bind different flows to the same originating port and the same destination port. When different flows are bound to the same originating ports but different destination ports, it means that the value at the originating ports is routed to multiple destination ports simultaneously. A produced material in the producer plan is copied to all the flows emanating from the producer.

A flow is annotated with a condition expression that is evaluated either true or false during the plan execution phase. The producer plan routes its value from the producer's corresponding port to the corresponding port of the consumer plan following the direction of the flow, if and only if the condition is evaluated as true. On the other hand, when the conditional is evaluated as false, the value of producer's corresponding port is not routed

to the consumer. This conditional routing mechanism is important in achieving data flow and control flow in our plan execution.

With this conditional routing mechanism, we can express the semantic of disjunction (OR), conjunction (AND) and exclusive disjunction (XOR). In Figure 11, binding two different producers to the same input port of a consumer embodies the semantics of XOR. The plan of "frying ham" and the plan of "frying bacon" are exclusive in the sense that no matter which one sends the data to the plan of "fry" first, the "fry" will initiate executing a plan. Because we do not allow re-execute a plan that is already completed, when a user gets ham, it erases the possibility of "getting bacon". A user could execute both "getting ham" and "getting bacon", however he can only complete fry once using either ham or bacon (but not both).

In the second case, conjunction, the output of the plan "getting ham" is bound to the input port of the plan "fry". The output of "getting bacon" is bound to a different input port of "fry". The plan "fry" can start executing only when both of its inputs are ready.

In the third case, disjunction, the output of the plan "getting ham" is bound to the input port of the "fry". On the other hand, the output of the plan "getting bacon" is bound to the input port of a different "fry" plan. In addition, those two plans are wrapped by a hierarchical plan called "breakfast". Thus, we can represent disjunction semantics with separate courses of actions included in a hierarchical plan.

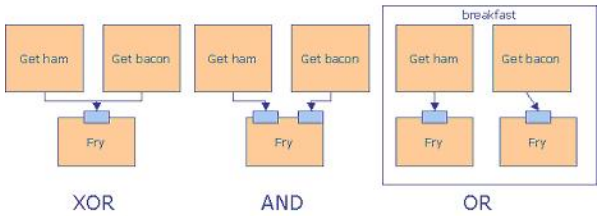

Figure 11 Representation of XOR, AND, OR with conditional data flow.

5.4 Semantics of Multi-flow

A multi-flow can represent multiple relationships between a producer and a consumer. A multi-flow can be instantiated as multiple distinct flows that map one producer and multiple consumers. Or it can be instantiated as multiple flows that map multiple producers and multiple consumers.

The two different instantiations are distinguished within context. If the multi-flow is defined between a multi-plan and a non-multi-plan like in Figure 12, then we treat the instantiation as the first case.

If the multi-flow is defined between multi-plans as in Figure 13, we treat the instantiation as the second case. In this case, the cardinality of instances in producer hierarchical plan should be the same as the one in consumer hierarchical plan.

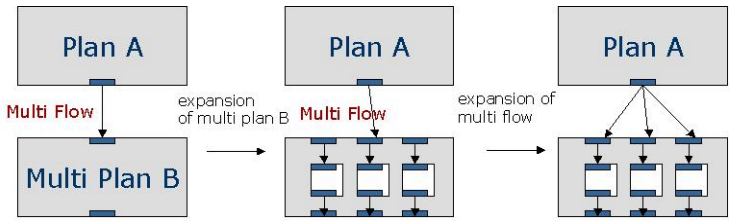

Figure 12 Expansion of Multi Flow between a producer plan and consumer Multi plan.

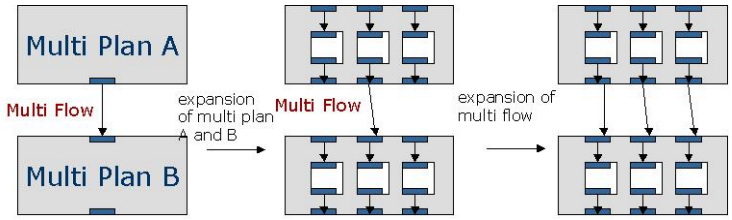

Figure 13 Expansion of Multi Flow between a producer Multi plan and consumer Multi plan.

The multi-flow definition cannot be used between a multi-plan and the plan that contains the multi-plan. The reasoning behind this design decision is that we do not propagate the effect of transformation to the container plan of a multi-plan.

5.5 State of a Plan

Plan instances during execution change state. There are 4 different kinds of state for plan instances.

The first state, "Waiting" describes a plan has not been executed yet and is not ready to be executed. The second state, "Candidate" means that a plan has not been executed yet and the plan is ready to be executed. The third state, "Done" means that the user explicitly indicated that the plan was completed. The fourth state, "Eliminated" means that the plan instances are eliminated from a set of next possible actions. In other words, they are eliminated from the set of instances with "Candidate" status without the user's explicit indication.

When a user points to a certain instance and enforces the completion of that instance via a hard complete, Planlet conforms to the user's intention by obediently updating the state of any affected instances. However, Planlet updates the state of instances in different ways. When a user explicitly confirms the completion of a certain step, the state is recorded as "Done". On the other hand, when Planlet derives the completion state of a certain step from producer-consumer relationships among plan instances, Planlet marks it as "Eliminated". When a step is notified to be complete by a user, the instances that are upstream of the step will be marked as "Eliminated".

For a step instance, the waiting state means that the plan is waiting for all of its input to be filled. Thus the step instances are identified as a waiting state if and only if there exists some input that is not ready. The "Candidate" state for a step instance means that this instance is possible to complete because inputs are ready and the ancestors of this instance are either completed by a user or believed to be completed from user's indication. Thus, the step instances with "Candidate" state are a set of next possible actions that a

user can perform. Step instances are in the "Done" state if and only if their outputs are produced and are indicated as produce by explicit user completion notification. The Step instances are in the "Eliminated" state if and only if all their outputs are produced and this fact is inferred from the plan graph, not explicitly indicated by the user.

For the other types of plan instances, they are identified as being in the waiting state if and only if there exists some sub plan in the waiting state and all sub plans are not in the candidate state. They are in the "Candidate" state if and only if there exists some sub plan in the "Candidate" state. They are in the "Done" state if and only if all children are in the "Done" state. Lastly, they are identified as "Eliminated" if and only if all children are either in the "Done" state or "Eliminated" state.

Semantically the "Eliminated" and "Done" state are the same except that the "Done" state is determined by the user's explicit completion indication and "Eliminated" is not.

5.6 Example: Preparing bacon and egg breakfast

Plan Definition

Figure 14 shows the plan definition for cooking a breakfast. The plan "Breakfast" consists of two sub plans, one is the plan "Bacon" to cook bacon, and the other is the plan "Egg" to cook eggs. In the plan definition, producer-consumer partial ordering relationships are shown.

The property "serving" is not initialized in the plan definition and the sub plans are parameterized with serving size. The value of serving is not bound to any value yet. Between the "Bacon" plan and "Egg" plan, there is no partial ordering enforced by a producer-consumer relationship.

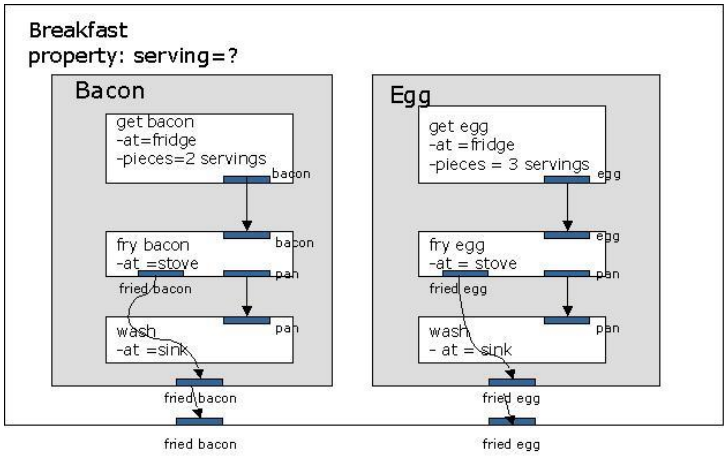

Figure 14 Plan Definition: Breakfast.

Plan Instantiation

Figure 15 shows the plan instantiated by a user interaction. When a user assigns the number of servings to a plan, the plan assigns the correct values for properties in the sub plans. For example, the "pieces" of "Bacon" is now initialized with 4, which is 2 times the number of servings.

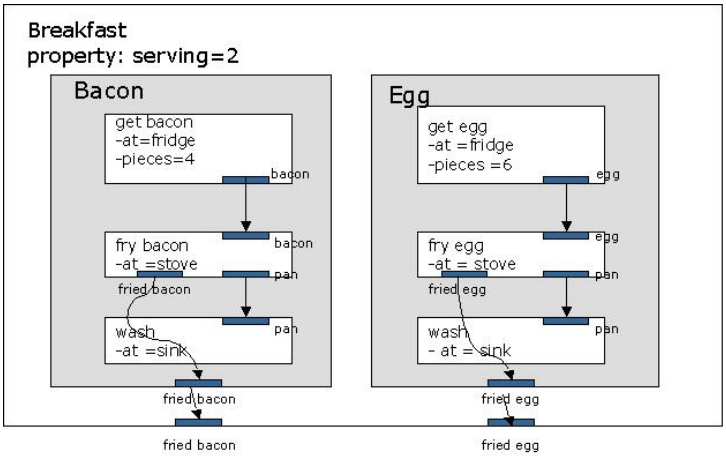

Figure 15 Plan Instantiation: Breakfast.

Plan Execution

There is many different ways of executing "Breakfast" plan. For example, a graduate student may cook bacon and eggs by cooking bacon completely and then starting to cook egg (Figure 16). A fast food chain employee (Figure 17) may cook bacon and eggs in the pan at the same time and then wash the pan. We have to notice that nothing prevents a fast food chain employee from cooking bacon and eggs at the same time given the producer–consumer relationships in the plan instance. On the other hand, a homemaker (Figure 18) may cook the bacon, wash the pan, cook the eggs and then wash the pan a second time. Given the many different ways of cooking breakfast, it's not sensible to enforce on the user a particular process to cook breakfast. However it is possible to assist a user by suggesting next possible actions based on the information acquired by interaction history and partial ordering relationships in the plan instance.

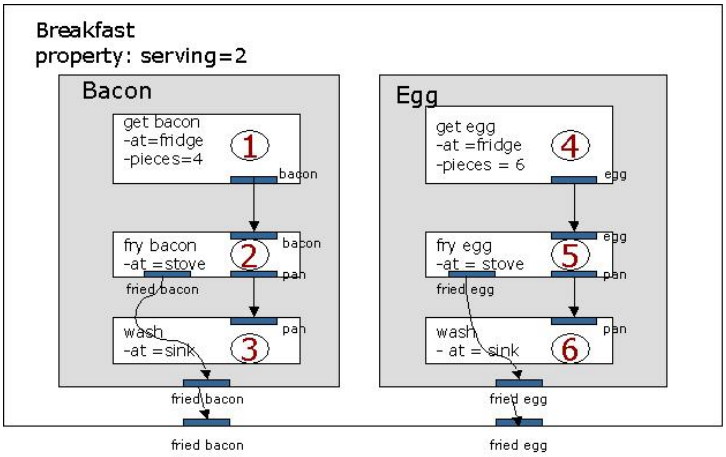

Figure 16 Breakfast Plan execution by a Graduate Student.

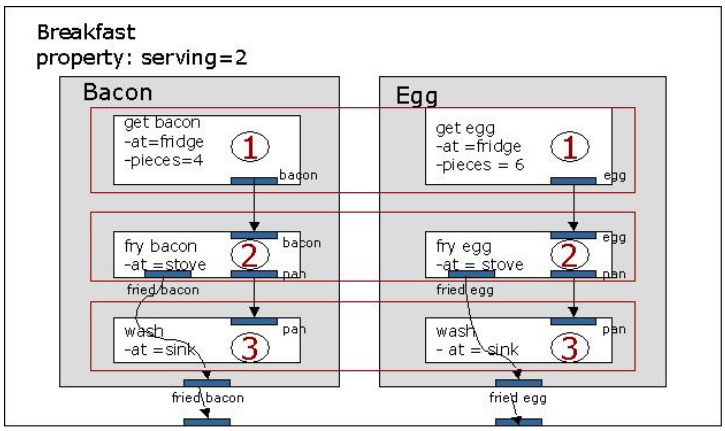

Figure 17 Breakfast Plan execution by a Fast Food Chain Employee.

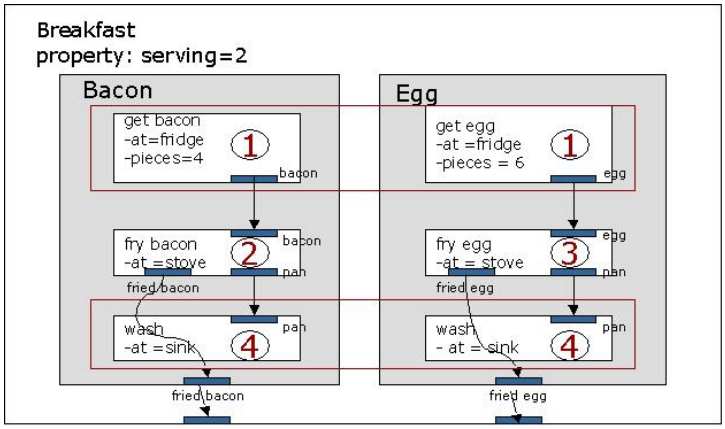

Figure 18 Breakfast Plan execution by a Housewife.

6 Implementing Planlet

This section describes our implementation of Planlet. In this section, we focus on the most important aspects for people who want to write Ubicomp applications on top of Planlet.

As mentioned in the previous section, applications define user plans using the Step, HierarchicalPlan, MultiPlan, LoopPlan, Flow, and MultiFlow constructs. The first four of these constructs also have corresponding Instances (e.g., StepInstances, HierarhcicalPlanInstances, etc) that are created during plan instantiation. Planlet implements the above as Java classes; the Planlet class hierarchy is shown in Figure 19.

We have merged the Plan hierarchy with the Instance hierarchy because Java does not support multiple inheritance. Ideally, we would have had Step, HierarchicalPlan, MultiPlan, and LoopPlan be direct descendants of Plan. Then, when creating each of the particular instance classes (e.g., LoopPlanInstance), we could have made them direct descendants of the corresponding Plan subclass (e.g., LoopPlan) *and* a second class (the Instance class).

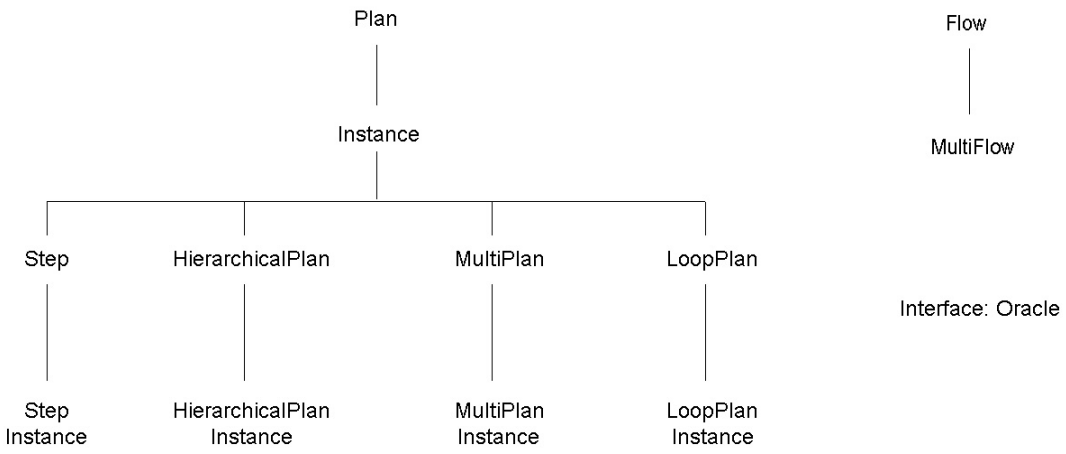

Figure 19: Planlet class hierarchy.

Given that we did not have multiple inheritance available to us, we chose to merge the hierarchies as shown. We felt this was better than making Instance an interface and then requiring each of the instance classes to re-`implement` the same code.

Once a plan is defined, it can be instantiated by calling its `instantiate(Oracle)` method. The Oracle is a Java interface with only one method – `getValue(valueName)`. Planlet uses the Oracle's `getValue` method to acquire the parameters it needs to instantiate a plan from the plan definition. The application using Planlet `implements` the Oracle as it sees fit; it could use a file with name-value pairs as the backend info source for the `getValue` method, it could prompt the user for the desired `valueName`, or it could use some other method to provide the data needed to instantiate a plan.

Calling `instantiate` on a plan definition returns an instance of the corresponding type (for example, `HierarchicalPlan.instantiate(oracle)` returns a HierarchicalPlanInstance).

Plan instances can be in one of four states: Candidate, Waiting, Done, or Eliminated. StepInstances and HierarchicalPlanInstances can be in any four of these states. For reasons we will explain below, LoopPlanInstances and –MultiPlanInstances can be in any state except Candidate. Planlet uses these states to inform the application about what actions (i.e., StepInstances) a user could be expected to perform next.

If a plan instance is in the Candidate state, that indicates a user can begin performing that part of the plan. In the case of a StepInstance this means that the plan instance has all the inputs it needs to be carried out. In the case of HierarchicalPlanInstance, this means that at least one of the plan instances contained within has all the inputs it needs to be carried out. LoopPlanInstances and MultiPlanInstances cannot be in the Candidate state because they are expanded into HierarchicalPlanInstances once their inputs are available (however, they can be in any of the other three states).

Plan instances in the Waiting state are waiting for input from some other plan to be produced before they can be performed. A plan instance that is Done has been marked completed by the user and all the outputs from that instance are (presumed) available for use as input by other plans.

```
softComplete() returns diagnostic {
  switch(state)
  case: Done
    throw Exception;  // the user is trying to complete something he has
                      // already completed; this action is not reflected
   // in the plan definition
  case: Eliminated
    state = Done;     // Planlet accepts marking an Eliminated plan as
                      // Done because it could be that the user is just
                      // telling the application of its actions in an
                      // out-of-order manner
    return Ok;
  case: Candidate
    hardComplete();   // all needed inputs are available, go ahead and
                     // mark this plan as complete
    return Ok;
  case: Waiting
    diagnostic := generateDiagnosis();  // this function determines if
                      // the plan is missing one or more inputs or,in the
                      // case of a HierarchicalPlanInstance, one or more
                      // subplans have not been completed
    return diagnostic;
}
```

Figure 20: Pseudocode for the `softComplete()` method.

Lastly, since Planlet does not assume or rely on a 100% up-to-date replica of the state of materials in the real world, it may be the case that the user indicates he is Done with a Step that Planlet thinks is currently Waiting. Any ancestors of this plan that are not marked done are then marked as Eliminated. This prevents them from being considered by Planlet as next possible actions by the user (since the Step just marked Done would, according to the plan, only be performed *after* the things just marked as Eliminated were performed). For auditing purposes, though Planlet doesn't consider them as next possible actions, it also doesn't consider them as Done because the application made no formal record of these actions as being Done.

Planlet offers applications two modes of marking a step as done: `softComplete` and `hardComplete`. When `softComplete` is called on a step or subplan, Planlet checks if that step is on the list of steps that the user is expected to take next. If not, it returns a diagnostic and does not change the state of the plan. Such diagnostic is examined by the application and an appropriate measure can be taken. In case the application is able to double check that it is ok to enforce the completion notice, it can call `hardComplete` on that same step or subplan. For a `hardComplete`, Planlet advances the plan execution state to the indicated point. Pseudocode for these two methods is provided in Figures 20 and 21.

The `markInputReady` and `markOutputReady` methods used in `hardComplete()` are helper functions that recursively trace the Flows going into each of the plan's inputs and coming from each of the plan's outputs, updating the state of the plans on the other end of the Flow according to the state descriptions given above.

```
hardComplete() {
  for all inputs, i, to this plan
    markInputReady(i);

  for all outputs, o, to this plan
    markOutputReady(o);

  if(this plan instance is a HierarchicalPlanInstance)
    for each subplan, sp, in this HierarchicalPlanInstance
      sp.hardComplete();

  state = Done;

  if(this plan is itself a subplan of a larger HierarchicalPlanInstance)
    updateState of larger HierarchicalPlanInstance
}
```

Figure 21: Pseudocode for the `hardComplete()` method.

To find out what Steps a user may be doing next for a given plan instance, an application can call the `getPossibleActions()` method. This method returns a list of Steps in the plan that are in the Candidate state. The plan builds up the list of next possible actions by recursively asking each of its subplans what their next possible actions are. The recursion stops at the Step level; a Step returns itself as a possible next action if it is in the Candidate state, null otherwise. Pseudocode for this method is shown in Figure 22.

```
getPossibleActions()returns Set {
  if(this plan is a StepInstance)
    if(this StepInstance is a Candidate)
      return {this Step}
    else
      return ∅

  else if(this plan is a HierarchicalPlanInstance)
    Set S = ∅
    for each subplan sp in this HierarchicalPlanInstance
      S.union(sp.getPossibleActions())
    return S

  // we need this last line of code because there may be some
  // LoopPlans or MultiPlans that haven't been expanded yet
  else
    return ∅
}
```

Figure 22: Pseudocode for the `getPossibleActions()` method.

References

1. Aho A., Sethi R., Ullman J., *Compilers. Principles, Techniques and Tools.* Addison-Wesley, 1986.
2. Antifakos S., Michahelles F., Schiele B., Proactive Instructions for Furniture Assembly, *UbiComp 2002: Ubiquitous Computing, Proceedings of the 4th International Conference*, Göteborg, Sweden, LNCS 2498, pp 351-360, September 2002
3. Arnstein L. F., Sigurdsson S., Franza R., Ubiquitous Computing in the Biology Laboratory, *Journal of Lab Automation* (JALA). 6(1), March 2001.
4. Garlan D., Siewiorek D., Smailagic A., Steenkiste P., Project Aura: Towards Distraction-Free Pervasive Computing. IEEE Pervasive Computing, 1(2):22--31, April-June 2002.
5. Georgakopoulos D., Hornick M., Sheth A., An Overview of Workflow Management: From Process Modelling to Workflow Automation Infrastructure, *Journal of Distributed and Parallel Databases*, 3(2), 1995.
6. Hecht M.S., Flow Analysis of Computer Programs. Elsevier, Amsterdam, 1977.
7. Kautz H., Arnstein L., Borriello G., Etzioni O., Fox D., An Overview of the Assisted Cognition Project, AAAI-2002 Workshop on Automation as Caregiver: The Role of Intelligent Technology in Elder Care, http://www.cs.washington.edu/homes/kautz/papers/.
8. Rich C., Sidner C.L., Lesh N.B., "COLLAGEN: Applying Collaborative Discourse Theory to Human-Computer Interaction", Artificial Intelligence Magazine, 22(4) pp 15-25, winter 2001.
9. Siewiorek D., Smailagic A., Bass L., Siegel J., Martin R., Bennington B., Adtranz: A Mobile Computing System for Maintenance and Collaboration, Proc. IEEE International Conference on Wearable Computers, Pittsburgh, PA, October 1998.
10. Weld D. S., An Introduction to Least Commitment Planning. AI Magazine, 15(4), pp 27–61, 1994.
11. Weld D. S., Recent Advances in AI Planning. AI Magazine, 20(2), pp 93–123, 1999.